# DPLL(Agg): an efficient SMT module for aggregates.

Broes De Cat and Marc Denecker
Department of Computer Science, K.U.Leuven, Belgium
{Broes.Decat, Marc.Denecker}@cs.kuleuven.be

K.U. Leuven

**Abstract** The use of aggregates often allow for a compact and natural encoding of many real-life problems. FO(Agg) is an extension of first order logic (FO) with aggregates. In this paper, we present algorithms for a satisfiability checking module for aggregate expressions in the context of the DPLL(T) architecture, achieving bound consistency. We consider among others cardinality, sum and maximum aggregates. The module can be used in all DPLL-based SAT, SMT and ASP solvers. The algorithms have a low complexity. We report on the incorporation of the algorithms in Minisat and the IDP-system, including an experimental evaluation.

## 1 Introduction

Knowledge representation (KR) is centered around representing human knowledge and using this knowledge to solve computational problems by applying inference. This makes KR an important discipline of AI and even of computer science in general. For KR paradigms to be broadly applicable, they should be both very expressive and possess tractable inference methods. Currently, methods being developed within domains like SAT, CP and ASP are starting to fulfill this promise: light-weight inference methods like finite model expansion are relatively cheap, while able to support large subsets of first-order logic or even more expressive languages.

One of the important concepts in any of those paradigms are aggregates functions: functions like sum, maximum or average which map a sets of elements to another value. Many application domains can be represented naturally using aggregates, for example planning, configuration and optimization problems. Constraint programming languages has a long history of research into aggregates, often called constraints on set, [1]. Within the field of logic programming, extensions to aggregates have been discussed by a.o. Van Gelder [15], Niëmela et al. [10] and Pelov et al.[12].

In this paper we introduce FO(Agg), the extension of first-order logic (FO) with general aggregates, and we show interesting properties like monotonicity and how to transform aggregates into a monotone form. We then present bound consistent propagation mechanisms for FO(Agg) and how to extend model

expansion for finite domain theories from FO to FO(Agg). For this we use the ground-and-solve methodology: the FO(Agg) theory is first *grounded* to its ground normal format $AggNF$, details of which can be found in [16]. For the solve step, we present algorithms for satisfiability checking of PC(Agg) theories in the context of the *DPLL(T)* architecture, an architecture which allows to extend satisfiability solvers for propositional logic with background theories $T$ [11].

We also show how to implement these algorithms as a concrete module within the DPLL($T$) framework for efficiently checking the satisfiability of aggregates. We present experiments using our finite model expansion system IDP for the language FO(.): FO extended with among others aggregates, inductive definitions, partial functions and a type hierarchy.

In section 2, we introduce FO and FO(Agg). Section 3 presents the ground format $AggNF$. Algorithms for constructing and implementing the aggregate module are given in 4 and 5. Sections 6 and 7 present optimizations and experiments. Related work and conclusions are discussed in 8 and 9.

## 2 Preliminaries

### 2.1 FO

We assume familiarity with classical first order logic (FO).

A vocabulary $\Sigma$ consists of a set of predicate and function symbols. Each predicate and function has an associated *arity* $n \in \mathbb{N}$. Propositional symbols and constants are 0-ary predicate, respectively function symbols. We assume that each vocabulary includes the equality symbol $= /2$.

A *term* over $\Sigma$ is defined as usual by induction over variables and constant and function symbols. Tuples of terms are denoted by $\bar{t}_1, \bar{t}_2, \ldots$, tuples of variables by $\overline{x}, \overline{y}, \ldots$. An *atom* is a formula of the form $P(\bar{t})$. A *literal* is an atom or its negation. An atom is usually denoted by $a$, a literal by $l$. A first-order logic formula over $\Sigma$ is defined as usual by induction over atoms, logical connectives ($\wedge$, $\vee$, $\neg$) and quantifiers ($\forall$, $\exists$). A variable $x$ is a free variable of formula $\psi$ if it occurs outside every quantified subformula $\exists x\varphi$ or $\forall x\varphi$. For a formula $\varphi$, we use $\varphi[\overline{x}]$ to indicate that $\overline{x}$ are its free variables. A *sentence* is a formula without free variables. For a variable tuple $\overline{x}$ and a term tuple $\bar{t}$ (of the same length), $\varphi[\overline{x}/\bar{t}]$ denotes the formula $\varphi$ in which all free occurrences of $\overline{x}$ have been replaced by $\bar{t}$. An occurrence of a subformula $\varphi$ in formula $\psi$ is *positive* (*negative*) if it occurs in the scope of an even (odd) number of negations. A *theory* is a finite set of sentences. A $\Sigma$-structure $I$ consists of a domain $D$ and

- a function $P^I : D^n \to \{\mathbf{t}, \mathbf{f}\}$ for each predicate symbol $P/n \in \Sigma$.
- a function $F^I : D^n \to D$ for each function symbol $F/n \in \Sigma$.

The value $t^I$ of a term $t$ in an interpretation $I$ and the satisfaction relation $\models$ are defined as usual (e.g. [5]). An interpretation $I$ is called a *model* of the formula $\varphi$ iff $I \models \varphi$. Interpretations $I[x : \mathbf{t}]$ and $I[x : \mathbf{f}]$ denote the extension of interpretation $I$ which interprets $x$ as true, respectively false.

### 2.2 FO(Agg) : adding aggregates to FO

To extend FO with aggregates, we follow the approach in [12]. An aggregate is an interpreted second order function which takes a set as an argument. Common aggregates are cardinality, summation, minimum, maximum, average, etc. A set expression is of the form $\{\overline{x} : \varphi[\overline{x}, \overline{y}]\}$ and is a symbolic description of the set of all tuples $\overline{x}$ that satisfy the formula $\varphi[\overline{x}, \overline{y}]$, for arbitrary but fixed values for the variables $\overline{y}$. The inductive definition of an FO(Agg) term and FO(Agg) formula extends the one of FO terms and formulas with an extra case defining $Agg(\{\overline{x} : \varphi[\overline{x}, \overline{y}]\})$ as FO(Agg) term if $Agg$ is an aggregate function and $\varphi[\overline{x}, \overline{y}]$ an FO(Agg) formula. Note that this allows for nested occurrences of aggregates. The semantics of FO(Agg) is defined by adding extra inductive rules for defining the value of set expressions $\{\overline{x} : \varphi\}$ and aggregate terms $Agg(\{\overline{x} : \varphi\})$.

*Example 1 (Power-unit maintenance).* In the following example, we want to express that the sum of capacities of power units not in maintenance per week exceeds some minimum threshold.

$$\forall w(sum\{(c, u) : Capac(u, c) \wedge \neg InMaint(u, w)\} \geq Threshold)$$

In this expression, the first argument of the tuples are summed up. This binary set expression basically represents the *multi-set* of the capacities of different units not in maintenance.

A technical issue with the aggregates considered is their value on the empty set. These are defined in the following table:

| $card(\emptyset)$ | $sum(\emptyset)$ | $prod(\emptyset)$ | $min(\emptyset)$ | $max(\emptyset)$ |
|---|---|---|---|---|
| 0 | 0 | 1 | $+\infty$ | $-\infty$ |

In the IDP-system, FO(Agg,...) sentences are *grounded* in the context of a given finite domain $D$ and brought into a normal form using the grounder *gidl* [17,16]. In the next section, we define this ground normal form for FO(Agg). For the sake of simplicity, we will restrict ourselves to the aggregates $sum, product$ and $maximum$. Each is a mapping from (multi-)sets of integer values to integers. Two other important aggregates, cardinality # and $minimum$ can be directly formulated in terms of these:

- $\#\{\overline{x} : \varphi\} = sum\{(w, \overline{x}) : w = 1 \wedge \varphi\}$
- $min\{x : \varphi\} = -1 \times max\{y : \exists x(y = -1 \times x \wedge \varphi)\}$

## 3 A ground normal format for aggregate expressions

In this section, we define the *Aggregate normal form*, a ground normal form based on CNF. A suitable ground format for representing (multi-)set expressions in the context of a given domain $D$ is through *weighted sets*. A weighted set $S$ is a set of tuples $(w_i, l_i)$ of ground literals $l_i$ and a value $w_i$. A weighted set is a symbolic

description of the multi-set of values $w_i$ for each $i$ such that $l_i$ holds. Since all aggregates considered in this paper range over numerical (multi-)sets, any $w_i$ will be a numerical value, called the *weight* of the tuple $(w_i, l_i)$. In the rest of the paper, a set will always denote a weighted set and a set literal will denote a literal in a weighted set.

**Definition 1 (Aggregate expression).** *A* ground aggregate atom *is an expression of the form $lwr \leq Agg(S)$ or $Agg(S) \leq upr$, with numerical values $lwr$ and $upr$ and a set $S$. The value $lwr$ is called the* lower bound*, $upr$ an* upper bound*. An* aggregate sentence *is of the form $P \equiv lwr \leq Agg(S)$ and $P \equiv Agg(S) \leq upr$, where $P$ is an atom called the* head *of the aggregate sentence. A theory in* Aggregate normal form *(AggNF) consists of ground clauses without aggregate atoms and a set of aggregate sentences.*

Grounding FO(Agg) theories in the context of a domain $D$ involves rewriting steps such as $P(\dots Agg\{\overline{x} : \varphi\} \dots)$ into $\exists x(P(\dots x \dots) \wedge x = Agg\{\overline{x} : \varphi\})$, rewriting equalities with aggregate terms into pairs of inequalities of aggregate atoms and reification of aggregate atoms (as expressed by aggregate sentences). A further problem is that such theories contain numerical expressions and hence $D$ includes the integers and is infinite, which may lead to infinite ground theories and infinitary ground clauses. Extra syntactic restrictions such as finite bounds on integer expressions have to be imposed to avoid this. We refer to [16] for more information on the grounding procedure. From now on, we assume that we have given a finite theory in $AggNF$.

## 4 Constructing an SMT module for aggregates

To support aggregates, we follow the DPLL($T$) architecture of SAT modulo Theories (SMT) [11]. A DPLL($T$) solver essentially builds a sequence of more and more precise partial interpretations, evaluates formulas in such interpretations, and computes propagations and conflicts. The kernel of this type of solver consists of a DPLL-based solver with unit propagation and clause learning [8] operating on a set of ground clauses. This kernel is combined with a set of $T$-solvers: constraint solvers/decision procedures which perform consistency checks and propagation for the current partial interpretation $I$ with respect to a background theory $T$.

Typically, background theories are implicit first- or second-order theories (e.g., Peano arithmetic). In our setting however, a theory $T$ is a singleton theory

$$\{P \equiv \mathcal{A}\}$$

consisting of a reification sentence for a ground aggregate atom: $\mathcal{A}$ is an atomic expression involving an aggregate and a weighted set, for example the expression $lwr \leq Agg\{(w, 1, l_1), \dots, (w_n, l_n)\}$ and $P$ its reification. In such a theory, propagation is to do one of the following things:

1. (Upward propagation) If $P$ is unknown in the current partial interpretation $I$ and $\mathcal{A}$ is true no matter how the unknown set literals $l_i$ are interpreted, then propagation derives that $P$ is true. Similarly if $\mathcal{A}$ is necessarily false.
2. (Downward propagation) If $P$ is true and a literal $l$, for which $l$ or $\neg l$ occur in the weighted set, is unknown in $I$, and if $\mathcal{A}$ would be necessarily false in $I$ extended with $\neg l$, then propagation derives $l$. The situation when $P$ is made false is analogous.

The propagation idea is obvious, but the problem is that in general, to compute propagations in an optimal way is computationally hard. E.g., consider the following aggregate atom $sum\{(w_1, l_1), \ldots, (w_n, l_n)\} = n$, with $n$ some integer value. If all literals $l_i$ are unknown, then the problem of determining whether $sum\{(w_1, l_1), \ldots, (w_n, l_n)\} = n$ is necessarily false corresponds to the well-known co-NP complete problem of deciding whether $\{w_1, \ldots, w_n\}$ has no sub(multi)set whose sum is $n$. This means that optimal upward propagation is NP-complete here. There is however an interesting class of aggregate atoms for which propagation can be computed in polynomial time.

We say a literal $l$ is *monotone* with respect to an aggregate atom $\mathcal{A}$ if increasing $l$'s truth value leads to increased truth value of $\mathcal{A}$. Formally, $l$ is monotone w.r.t. $\mathcal{A}$ if for each two-valued interpretation $I$, $\mathcal{A}^{I[l:\mathbf{f}]} \leq_t \mathcal{A}^{I[l:\mathbf{t}]}$; $l$ is anti-monotone if for each two-valued interpretation $I$, $\mathcal{A}^{I[l:\mathbf{f}]} \geq_t \mathcal{A}^{I[l:\mathbf{t}]}$. A class of aggregate atoms with polynomially computable propagation are those in which each set literal is either monotone or anti-monotone. Indeed, for a given partial interpretation $I$ and an aggregate atom $\mathcal{A}$, let $M$ consist of all unknown monotone set literals and $A$ of all unknown anti-monotone ones. Then $\mathcal{A}^{I[M:\mathbf{t},A:\mathbf{f}]}$ is an upper bound and $\mathcal{A}^{I[M:\mathbf{f},A:\mathbf{t}]}$ a lower bound to the truth value of $\mathcal{A}$ in any interpretation extending $I$. Hence, if this lower bound is true or the upper bound is false, then upward propagation can be performed deriving $P$ to be true, respectively false. Also, if $P$ is true and for the monotone set literal $l$, $\mathcal{A}^{I[l:\mathbf{f},M\setminus\{l\}:\mathbf{t},A:\mathbf{f}]}$ is false then by downward propagation $l$ can be inferred to be true. Other downward propagation is possible for anti-monotone literals and for when $P$ is false. All these propagations can be computed in polynomial time, provided that the value of the aggregate atom in a two-valued interpretation is polynomially computable. We define:

$$Min^I_{Agg(\mathcal{A})} = \mathcal{A}^{I[M:\mathbf{f},A:\mathbf{t}]} = min\{\mathcal{A}^J | J \geq_p I, J \text{ is two-valued}\} \quad (1)$$

$$Max^I_{Agg(\mathcal{A})} = \mathcal{A}^{I[M:\mathbf{t},A:\mathbf{f}]} = max\{\mathcal{A}^J | J \geq_p I, J \text{ is two-valued}\} \quad (2)$$

Here $J \geq_p I$ means that $J$ is more precise than $I$, i.e., $J$ can be obtained from $I$ by setting unknown literals to true or false.

**Proposition 1.** *Assume that in a set $S = \{(w_1, l_1), \ldots, (w_n, l_n)\}$, each literal $l_i$ occurs only once, and its negation does not occur in $S$. In that case, each $l_i$ is either monotone or anti-monotone with respect to the aggregate expressions $sum(S) \leq upr$, $lwr \leq sum(S)$, $prod(S) \leq upr$, $lwr \leq prod(S)$, $max(S) \leq upr$ and $lwr \leq max(S)$. Moreover,*

- *$l_i$ is monotone w.r.t $lwr \leq sum(S)$ iff $w_i$ is positive.*
- *$l_i$ is monotone w.r.t $sum(S) \leq upr$ iff $w_i$ is negative.*
- *$l_i$ is monotone w.r.t $lwr \leq max(S)$ and anti-monotone with respect to $max(S) \leq upr$.*
- *To obtain a similar property for product aggregates, we will limit the weights of product aggregates to $[1, +\infty)$. In that setting, $l_i$ is monotone w.r.t $lwr \leq prod(S)$ and anti-monotone with respect to $prod(S) \leq upr$.*

*Example 2.* Consider the aggregate sentence

$$P \equiv 3 \leq sum\{(-5, A), (-2, B), (0, C), (1, D), (9, E)\}$$

Literals $A$ and $B$ are anti-monotonous, $D$ and $E$ are monotone. Literal $C$ has no effect on the set and can be dropped.

Now assume an interpretation $I = \{\neg D, E\}$. The resulting lower bound is $Min^I_{Agg(\mathcal{A})} = \mathbf{t}$, the upper bound is $Max^I_{Agg(\mathcal{A})} = \mathbf{t}$. Consequently, $P$ can be derived to be true by upward propagation.

Secondly, consider an interpretation $I = \{P, A\}$. For literal $E$, $\mathcal{A}^{I[E:\mathbf{f}, M\setminus\{E\}:\mathbf{t}, A:\mathbf{f}]}$ is false, so downward propagation allows to derive $E$ has to be true.

One concern left is that the above proposition requires that a set should not contain complementary literals and only one occurrence per literal. The following proposition shows that specific aggregate atoms containing a set that does not satisfy this condition can easily be transformed in one that does.

**Proposition 2.** *Given a set $S = \{(w_1, l_1), \ldots, (w_n, l_n)\}$.*

- *For each atom $Q$ occurring positively or negatively in $S$, define $w_Q = \sum_{l_i = Q} w_i - \sum_{l_i = \neg Q} w_i$ and $D_Q = \sum_{l_i = \neg Q} w_i$. Define $D = \sum_{i=1}^{m} D_{Q_i}$. Then $sum(S) = sum\{(w_{Q_1}, Q_1), \ldots, (w_{Q_m}, Q_m)\} + D$ is true in each interpretation.*
- *For each atom $Q$ that occurs positively in $S$, define $w_Q$ as the maximum of the set $\{w_i | l_i = Q\}$. For each atom $Q$ occurring negatively in $S$, define $w_{\neg Q}$ as the maximum of the set $\{w_i | l_i = \neg Q\}$. Define $S_m$ as the set of all tuples $(w_Q, Q)$ such that $Q$ occurs positively in $S$, and $S_m^{\neg}$ as the set of all tuples $(w_{\neg Q}, \neg Q)$ such that $Q$ occurs negatively in $S$. Then $max(S) = max(max(S_m), max(S_m^{\neg}))$ is true in all structures.*

The case of the product aggregate is similar to that of sum.

*Example 3.* An aggregate atom $sum\{(-5, A), (-2, -A), (2, B), (5, B)\}$ can be reduced to $sum\{(-3, A), (7, B)\} - 2$.

An aggregate atom $max\{(-5, A), (-2, -A), (2, B), (5, B)\}$ can be reduced to $max\{max\{(-5, A), (5, B)\}, max\{(-2, -A)\}\}$.

In the case of each aggregate atom $\mathcal{A}$ considered in Proposition 1 and given a truth value for each weighted set literal $l_i$, the truth value of $\mathcal{A}$ can be computed in linear time in the size of the weighted set (if we assume constant time for summing and multiplying numbers). Thus, for these cases, propagation can

be performed in polynomial time. The propagation achieved is generally called *bound consistent*: for any aggregate atom $\mathcal{A}$, if for a partial interpretation $I$ no more propagation is possible on $\mathcal{A}$, then $I$ can be extended to a total interpretation such that the truth value of the head is consistent with the result of comparing the value of the set with the bound.

In an efficient implementation it is crucial to implement propagation in an incremental way. I.e., when the SAT solver derives a literal $l$ such that $l$ or its negation occurs in an aggregate sentence, then testing for upward or downward propagation through this sentences should be fast. In the next section, we introduce such methods. They are based on the following concepts. For each aggregate term $Agg(S)$ of the form $sum(S)$, $prod(S)$ and $max(S)$, we define two values:

$$Min^I_{Agg(S)} = min\{Agg(S)^J | J \geq_p I, J \text{ is two-valued}\} \tag{3}$$

$$Max^I_{Agg(S)} = max\{Agg(S)^J | J \geq_p I, J \text{ is two-valued}\} \tag{4}$$

The interest of these values is that, first of all, for an aggregate atom $\mathcal{A}$ involving $Agg$, $Min^I_{Agg(\mathcal{A})}$ and $Max^I_{Agg(\mathcal{A})}$ can be straightforwardly computed from $Min^I_{Agg(S)}$ and $Max^I_{Agg(S)}$, and second, that these values can be computed efficiently in an incremental way. For example:

- $Min^I_{Agg(Agg(S) \leq upr)}$ is $\mathbf{t}$ if $Max^I_{Agg(S)} \leq upr$ and $\mathbf{f}$ otherwise
- $Max^I_{Agg(Agg(S) \leq upr)}$ is $\mathbf{f}$ if $upr < Min^I_{Agg(S)}$ and $\mathbf{t}$ otherwise

Informally, this indicates that the aggregate atom is certainly true if the maximum value of the aggregate is below the upper-bound, and that the aggregate atom is certainly false if the minimum value is larger than the upper-bound.

As for the incremental computation of $Min^I_{Agg(S)}, Max^I_{Agg(S)}$, assume that $Min^I_{Agg(S)}$ and $Max^I_{Agg(S)}$ are given and an unknown literal $l$ is made true. In that case, we can efficiently compute the new values under $I[l : \mathbf{t}]$ according to the following table:

- $Min(sum(S)) := \begin{cases} Min(sum(S)) + w_i \text{ if } w_i \geq 0 \text{ and } l = l_i \\ Min(sum(S)) - w_i \text{ if } w_i < 0 \text{ and } l = \neg l_i \\ Min(sum(S)) \text{ otherwise} \end{cases}$
- $Max(sum(S)) := \begin{cases} Max(sum(S)) + w_i \text{ if } w_i < 0 \text{ and } l = l_i \\ Max(sum(S)) - w_i \text{ if } w_i \geq 0 \text{ and } l = \neg l_i \\ Max(sum(S)) \text{ otherwise} \end{cases}$
- $Min(max(S)) := \begin{cases} w_i \text{ if } w_i > Min(max(S)) \text{ and } l = l_i \\ Min(max(S)) \text{ otherwise} \end{cases}$
- $Max(max(S)) := \begin{cases} max(\{w_j | j \neq i\} \text{ if } Max(max(S)) = w_i \text{ and } l = \neg l_i \\ Min(max(S)) \text{ otherwise} \end{cases}$

Note that all operations are in constant time except for finding the maximum when the literal associated with the current maximum becomes false. Even in that case, complexity of finding the new maximum is amortized constant.

*Example 4.* Given an aggregate $sum(S)$, $S = \{(1, A), (-3, B), \ldots\}$ and $Min(sum(S)) = 15$.

– If $A$ becomes true, then $Min(sum(S)) = 16$.
– If $\neg A$ or $B$ become true, then $Min(sum(S)) = 15$.
– If $\neg B$ becomes true, then $Min(sum(S)) = 18$.

## 5 Implementing the SMT-interface.

In [11], an interface is presented which, when implemented by any theory solver, is sufficient to integrate the solver as a $T$-solver in DPLL($T$). Below, we discuss the implementation of the main operations of this interface for theory solvers for aggregate sentences. In our implementation, we create a module which takes care of propagation for any number of aggregate expressions, effectively aggregating any number of $Agg$-solvers within one module. But conceptually, they behave like separate $T$-solvers, because they are treated independently (no propagation between aggregates is done).

### 5.1 createTheory($T$)

When calling this procedure with an aggregate sentence, the sentence is brought into aggregate normal form, data-structures are created and the initial values for $Min_{Agg(S)}$ and $Max_{Agg(S)}$ are calculated. It is verified whether the aggregate atom is trivially entailed to be true or false. In that case, the head is propagated to the DPLL solver to be true or false and no $T$-solver is constructed.

### 5.2 setTrue($l$)

This is the main procedure and it is called when the literal $l$ is made true, i.e., upon the transition from partial interpretation $I$ in which $l$ is unknown to $I[l : \mathbf{t}]$. It returns the set of all propagations that can be made from $I[l : \mathbf{t}]$ and not from $I$. This may lead to upward or downward propagation for an aggregate sentence $P \equiv \mathcal{A}$ in case $l$ or $\neg l$ appears in it. Below, we discuss the different cases, they are presented in more detail in in [7]. Assume that $P \equiv \mathcal{A}$ involves an aggregate term $Agg(S)$. There are three main cases:

– $l$ is $P$ or $\neg P$. In this case, downward propagation is executed.
– $l$ or $\neg l$ is a weighted set literal of $S$. First we update $Min_{Agg(S)}$ and $Max_{Agg(S)}$ as described in the previous section. If these values do not change, then no propagation is possible. Otherwise, if $P$ is unknown then upward propagation is performed, otherwise downward propagation is performed.

**Upward propagation** When $P$ is unknown in $I$ and a weighted set literal $l$ is made true or false and this leads to a modification of $Min_{Agg(S)}$ or $Max_{Agg(S)}$, we check if $Min_{\mathcal{A}} = Max_{\mathcal{A}}$, and if that is the case, we derive that $P$ has the same value and propagate it.

**Downward propagation** This case occurs when a head atom $P$ or a weighted set literal $l$ or the negation of it was derived and this led to a different value of

$Min_{Agg(S)}$ or $Max_{Agg(S)}$. Both cases are treated exactly the same way. Let $I$ be the current updated partial interpretation (i.e., with $l$ set to its new value). The basic idea is simple:

- Let $P^I$ be true. For each unknown weighted set literal $l_i$, if $l_i$ is monotone with respect to $\mathcal{A}$ and $Max_{\mathcal{A}}^{I[l_i:\mathbf{f}]} = \mathbf{f}$, then propagate $l_i$; if $l_i$ is anti-monotone with respect to $\mathcal{A}$ and $Max_{\mathcal{A}}^{I[l_i:\mathbf{t}]} = \mathbf{f}$, then propagate $\neg l_i$.
- Let $P^I$ be false. For each unknown weighted set literal $l_i$, if $l_i$ is monotone with respect to $\mathcal{A}$ and $Min_{\mathcal{A}}^{I[l_i:\mathbf{t}]} = \mathbf{t}$, then propagate $\neg l_i$; if $l_i$ is anti-monotone with respect to $\mathcal{A}$ and $Max_{\mathcal{A}}^{I[l_i:\mathbf{f}]} = \mathbf{t}$, then propagate $l_i$.

Upward propagation involves one check for $P$ and downward propagation involves a linear number of checks, one for each unknown set literal $l_i$. As we saw, these checks can be done efficiently since the updated values of $Min_{Agg(S)}$, $Max_{Agg(S)}$, $Max_{\mathcal{A}}$ and $Max_{\mathcal{A}}$ can be computed efficiently.

The most expensive operation, downward propagation, can be further optimized by taking the monotonicity properties of aggregate expressions into account. We illustrate this in the case of the aggregate sentence $P \equiv lwr \leq sum(S)$. Assume that $P$ is true and consider the current value for $Max_{sum(S)}$. Any unknown literal $l$ which, when made true, would reduce $Max_{sum(S)}$ to strictly less than $lwr$ should be made false. What are these literals? They are all literals $\neg l_i$ with negative weights $w_i < lwr - Max_{sum(S)}$ and all literals $l_i$ with positive weights $w_i > Max_{sum(S)} - lwr$. By keeping the set sorted, using binary search on it and comparing with the previous values $lwr - Max_{sum(S)}$ and $Max_{sum(S)} - lwr$, these literals can be generated efficiently. Similar observation holds for when $P$ is false, for an upper-bounded sum and for $max$.

To find all occurrences of $atom(l)$ in aggregate sentences, a separate data structure is kept which maps a literal to all aggregate sentences in which it occurs (with specific information whether it occurs as a head, as a literal in the set or as its opposite). This data structure is initialized during createTheory. An additional data structure maps atoms to the aggregate sentences (if any) in which they occur in the head.

### 5.3 backtrack($n \in \mathbb{N}$)

Backtracking a propagated literal merely comes down to returning the data structures to the previous state. This can be done in constant time by storing the previous $Max_{Agg(S)}$ and $Min_{Agg(S)}$ values on a stack and using the already stored information of the partial interpretation $I$.

### 5.4 getExplanation($l$)

This procedure returns a clause, called the reason clause of $l$. One of the strengths of current SAT and SMT solvers is clause learning: when a conflict occurs, a clause, called the *learned clause* is created that constitutes an explanation for

the conflict in terms of literals on earlier decision levels. The learned clause allows to backtrack deeper, cutting out larger regions of the search tree. Also, it is itself a clause which can be added to the theory to prevent entering the same (sub)search space later on.

A requirement to construct the learned clause is that for each propagated literal $l$, the solver has to be able to return a *reason clause*. This is a clause that is entailed by the theory, and that was a unit clause with unit literal $l$ just before $l$ was derived (i.e., all literals except $l$ were false preceding the propagation of $l$). Reason clauses with few literals and old literals are preferred because they lead to deeper backtracking and are more useful for propagation at later stages of the search.

**Proposition 3.** *Assume an aggregate sentence with set $S$ and head $P$ and assume a current partial interpretation $I$. If a literal $l$ can be derived from the sentence, given $I$, then a reason clause for $l$ is the clause $(\bigvee_{l_1 \in L_1} l_1) \vee (\bigvee_{l_2 \in L_2} l_2) \vee l$, with:*

- *$L_1 = \{\neg P\}$, if $P$ is true in $I$. If the head is false in $I$, $L_1 = \{P\}$. Otherwise $L_1$ is empty.*
- *$L_2$ is the set of all literals which are true in $I$ of which the atom or its negation occurs in $S$.*

*Example 5.* Given an aggregate sentence $P \equiv sum(\{(1, A), (2, B)\}) \leq 1$ and a partial interpretation $I = \{P, A\}$. Then $\neg B$ can be derived by downward propagation, equivalent to unit-propagation on the reason clause $\neg P \vee \neg A \vee \neg B$.

If an earlier partial interpretation can be used, this might lead to smaller reason clauses, so the reason clause should be generated by the *earliest* interpretation from which $l$ could be derived. The partial interpretation at a previous moment can be easily looked up by storing for each literal the stack size at the moment of derivation.

In general it is NP-complete to find the minimal, earliest reason clause. In some cases however, we can do better: for maximum aggregates, $l \vee l_o$, with $l_o$ the last literal propagated before deriving $l$, is a sufficient, minimal reason clause in the following cases:

- When the aggregate is lower-bounded and $l$ is the head.
- When the aggregate is upper-bounded and $\neg l$ is the head.
- When $\neg l$ is in the set.

## 6 Optimizations

### 6.1 Set reuse

Experiments have shown that in some applications the same aggregate term $Agg(S)$ is reused in multiple aggregate sentences. The magic series problem is a good example of this: given a natural number $n$, find a function $f : [0, n -$

$1] \to [0, n]$ with the condition that $f(n)$ equals the number of integers $m$ such that $f(m) = n$. A straightforward[1] encoding of this problem in FO(Agg) is the formula $\forall x(Magic(x) = \#\{y|Magic(y) = x\})$, where $x$ and $y$ range over $[0..n-1]$. The grounding of this formula contains for each number $i$ in this range the following $2(n+1)$ aggregate sentences, for each $j \in [0..n]$:

$$p_{i,j} \equiv sum(\{(1, Magic(0) = i), \ldots, (1, Magic(n-1) = i)\} \leq j \quad (5)$$

$$q_{i,j} \equiv sum(\{(1, Magic(0) = i), \ldots, (1, Magic(n-1) = i)\} \geq j \quad (6)$$

such that $p_{i,j} \wedge q_{i,j}$ expresses that $\#\{k|Magic(k) = i\} = j$. All these aggregate sentences share the same weighted set expression.

The IDP grounder tries to detect multiple occurrences of the same ground aggregate term $Agg(S)$ in multiple aggregate atoms and if it succeeds, it creates only one representation for it which is shared over multiple aggregate atoms and sentences. This prevents redundant computations ($Min_{Agg(S)}$ and $Max_{Agg(S)}$ need to be computed only once instead of $2(n+1)$ times) and reduces memory overhead. Experiments in section 7 show that this is an important optimization.

### 6.2 Propagation by watched literals

In the propagation algorithm presented in Section 5, whenever an atom is derived of which the positive or negative literal occurs in the set $S$ of an aggregate term $Agg(S)$, the update process for $Min_{Agg(S)}$ and $Max_{Agg(S)}$ is called, potentially doing propagation. Obviously, starting the update process for every such derivation might sometimes create a lot of overhead. This can be partly avoided by using an alternative propagation scheme, comparable to the 2-watched literal scheme [9]. In such a scheme, a set of *watched* literals is maintained for any sentence $p \equiv \mathcal{A}$ of the ground theory. The vital property of such watched literal sets is that, as long as none of those literals becomes false, the sentence will not be false. For our aggregate expressions, we will maintain two sets of *watched* literals for any aggregate sentence $p \equiv \mathcal{A}$: one for $p \Rightarrow \mathcal{A}$ and one for $p \Leftarrow \mathcal{A}$.

Below, we illustrate this idea for $P \equiv lwr \leq \#(S)$ with $S = \{(w_1, l_1), \ldots, (w_n, l_n)\}$. Candidate watched literal sets for $P \Rightarrow lwr \leq \#(S)$ are any set consisting of $\neg P$ and $lwr$ non-false set literals or any set of $lwr + 1$ non-false set literals. If such a set is watched and one of its literals is derived to be false, then first an alternative watched literal set is searched for. If such a set exists, that set becomes watched. Otherwise, if no such set exists, propagation will certainly be applicable. E.g., if $lwr + 1$ set literals were watched and one is made false, then the remaining $lwr$ literals are propagated to be true. If $\neg P$ and $lwr$ set literals were watched and $P$ is made true, then the remaining $lwr$ set literals are made true. If it is one of the $lwr$ other literals that becomes false, then $\neg P$ is propagated. Candidate watched literal sets for $P \Leftarrow lwr \leq \#(S)$ consist of $P$ and $n - lwr + 1$ negations of set literals or $n - lwr + 2$ negations of set literals. Propagation is similar.

Combining these ideas shows for an expression $P \equiv lwr \leq \#(S)$, we need $n + 2$ watched literals, opposed to $2n + 2$ for the already presented algorithm.

[1] ←In fact, this problem is in P and is solved by a simple deterministic algorithm.

If $P$ would be known to be true, $P \Leftarrow lwr \leq \#(S)$ would be redundant, so only $lwr + 1$ watches are necessary. This situation frequently occurs when an aggregate expression occurs as a *global constraint*.

Such a watched literal scheme also has drawbacks. First, when a watched literal is becomes false, an alternative watched literal set has to be searched. This is a more expensive operation than the cheap updates of $Min_{Agg(S)}$ and $Max_{Agg(S)}$, because no incremental updates are possible (the whole aim of delaying propagation). Also, combination with the set reuse strategy is much more complicated: the possible watched literal sets depend on the bounds in the aggregate atom and on the head. There is clearly a trade-off here: if an aggregate expression does not occur often, and only few literals have to be watched, the watched literal scheme will be advantageous; otherwise the algorithm of the previous section will probably score better. It is part of future work to experiment with the watched literal scheme and compare it with the current algorithm.

### 6.3 Reduction of Maximum to SAT

Maximum aggregate sentences involving a set $S = \{(w_1, l_1), \ldots, (w_n, l_n)\}$ can be easily reduced to propositional logic using following transformation:

$$lwr \leq max(S) \equiv \bigvee_{lwr \leq w_i} l_i \qquad max(S) \leq upr \equiv \bigwedge_{upr < w_i} \neg l_i$$

In general this rewriting turns an expression into an even smaller propositional formula which is handled efficiently by SAT solvers. As shown following experiments, the gain of using specific algorithms for maximum aggregates is only significant when sets can be reused between aggregates.

## 7 Experiments

We experimentally evaluate our implementation for finite model expansion of FO(Agg) theories. We compare with finite model expansion in FO and with Answer Set Programming, using the ASP-system Gringo+Clasp, which is currently the best ASP-solver, as shown by the second answer set competition [3]. It has native support for aggregates.

The machine used was a dual-core 2.4 GHz with 4 Gb RAM, with Ubuntu 8.04 OS. We used the following solvers: gidl 1.6.6, Minisat(ID) 2.0b and Minisat(ID) 09z, Gringo 2.0.5 and Clasp 1.3.3. Minisat(ID) is a state-of-the-art SAT-solver, used in the IDP system. Minisat(ID) is the extension of the SAT-solver Minisat with support for inductive definitions [7]. Minisat 2.0b is the standard version, Minisat 09z is a newer version that won the Minisat Hack track of the SAT 2009 competition. We did experiments with four different setups:

**MA** gidl combined with minisat(ID)2.0b combined with our aggregate module.
**MZA** gidl combined with minisat(ID)09z combined with our aggregate module.
**MZ** gidl combined with minisat(ID)09z. All aggregates were eliminated by encoding them as FO formulas (using an "accumulator" argument, a common technique in Prolog programming).

**C** gringo+clasp.

The support for arbitrary size weights in our module (via the GNU Multi-Precision library) was turned off.

An important note about the experiments is that it is difficult to compare different solvers only on their aggregate performance: both encoding, grounder and supported ground format have a significant impact on the efficiency. Also, different SAT-solver heuristics have a large impact on the solver time (hence experiments with 2 SAT solvers for our module). Our aggregate module itself also interacts with the heuristics of the Minisat SAT-solver (it basically increases the priority of any literal propagated by the aggregate module).

We performed experiments with four different model expansion problems. We first concentrated on maximum aggregates, comparing efficiency between encoding them as FO and FO(Agg) and by grounding them from FO(Agg) to propositional logic. For this we used two simple theories:

$$T_1 = \forall a(max\{x : Distance(a,x)\} > a)$$
$$T_2 = \begin{cases} \forall x(P(x) \equiv max\{(n,a) : Distance(a,n)\} > x) \\ \exists ax(Distance(a,x)) \end{cases}$$

Set reuse is only possible for $T_2$, not for $T_1$. Predicate $Distance(N,D)$ maps nodes $N$ to distances $D$, $P(D)$ is a property on distances. We consider two instances $I_1 : D = [1 \dots 200], N = [1 \dots 20]$ and $I_2 : D = [1 \dots 500], N = [1 \dots 200]$. Secondly, we did experiments with the following model expansion problems, using cardinality and sum aggregates:

**Magic sequence** The magic sequence problem described in section 6.1, with $N$ the size of the domain.

**Social golfer** A planning problem aimed at partitioning $ps$ golfers into $g$ groups of fixed size $gs$ ($g \times gs = ps$) for $w$ weeks, such that:
- 2 golfers only meet each other at most one week.
- Each golfer plays in some group each week and all groups have size $gs$.

**Weight-bounded dominating-set** Given a weighted directed graph $G = (V, E)$ with $V$ a set of nodes and $E$ with weights $w_{(a,b)}$ for an edge $(a, b)$. Then, given a number of nodes $k$ and a minimum weight $w$, it is the problem of finding a subset $D$ of $V$, such that $\|D\| \leq k$ and, for each node $v \in V$, at least one of the following conditions holds:
- $v \in D$,
- $sum\{w_{(i,v)} | (i,v) \in E, i \in D\} \geq w$ or
- $sum\{w_{(v,j)} | (v,j) \in E, j \in D\} \geq w$.

For magic sequence, we experimented with the instances $N$=[10, 20, 30, 40, 50, 60, 70, 80, 90]. The social golfer problem was solved for all combinations of of 3, 4, 5 or 6 weeks, 3, 5, 7 or 9 groups and group sizes ranging from 3 to 6. For the weight-bounded dominating-set, we used the instances of the second answer set competition [2].

Tables 1 to 4 show the results of the experiments (in the order presented). In each table, we show timing results for the setups MZ, MZA, MA and C for the

| Theory | Instance | Groundsize in PC(Agg) in Mb | Groundsize in PC in Mb | time MZA | time MZ | time MZAasPC |
|---|---|---|---|---|---|---|
| 1 | $I_1$ | 0.03 | 2.17 | 0.07 | 0.88 | 0.12 |
| 1 | $I_2$ | 0.99 | 114.47 | 0.51 | 144.23 | 0.60 |
| 2 | $I_1$ | 0.06 | 2.6 | 0.38 | 1.07 | 0.58 |
| 2 | $I_2$ | 1.56 | 177.6 | 6.97 | 62.78 | 33.33 |

**Table 1.** Table comparing efficency and grounding size of encoding maximum aggregates as FO-formulas and as aggregates in FO(Agg).

| N | time MZ | time MA | time MZA | time C | N | time MZ | time MA | time MZA | time C |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 2.95 | 0.03 | 0.02 | 0 | 60 | 300 | 2.64 | 6.54 | 5.51 |
| 20 | 300 | 0.08 | 0.06 | 0.1 | 70 | 300 | 4.66 | 9.19 | 3.63 |
| 30 | 300 | 0.29 | 0.31 | 0.17 | 80 | 300 | 8.77 | 11.64 | 26.9 |
| 40 | 300 | 0.92 | 1.77 | 0.46 | 90 | 300 | 119.02 | 65.26 | 10.63 |
| 50 | 300 | 3.78 | 1.48 | 0.79 | | | | | |

**Table 2.** Results of model expansion for the magic sequence problem.

| Group-size | time MZ | time MA | time MZA | time C |
|---|---|---|---|---|
| 3 | 153.06 | 11.04 | 2.36 | 368.14 |
| 4 | 669.59 | 662.32 | 308.28 | 324.61 |
| 5 | 955.28 | 1094.76 | 950.55 | 331.74 |
| 6 | 2727.87 | 2995.52 | 2287.32 | 2124.51 |
| Time-outs: | 14 | 14 | 9 | 10 |

**Table 3.** Results of model expansion for the social golfer problem.

| Instance size | time MZ | time MA | time MZA | time C |
|---|---|---|---|---|
| $100x400$ | ### | 57.97 | 37.16 | 6.05 |
| $150x600$ | ### | 197.68 | 201.17 | 1243.14 |
| $200x800$ | ### | 182.01 | 321.2 | 1078.91 |
| Time-outs | ### | 3 | 2 | 7 |

**Table 4.** Results of model expansion for the weight-bounded dominating-set problem.

different instances, except for the experiment with maximum aggregates in which we use the setups MZA, MZ and an implementation of MZA which rewrites maximum aggregates as clauses (MZAasPC). For social golfer, we combined results of instances with the same group size and for weight-bounded dominating-set we combined results with the same domain size. All timing results are in seconds. In table 1, we also show the grounding sizes of using both an FO and an FO(Agg) encoding. All timing results are in seconds and grounding times are always included, the timeout was set at 300 seconds (### indicates timeout). For the last two experiments, we also show the number of time-outs. [2]

The experiments show on the one hand that natively supporting aggregates can increase the efficiency of model expansion and greatly decrease the size of groundings. For maximum aggregates, using a reduction to SAT is most as efficient as supporting maximum in the solver, while it is much less efficient when sets are reused. The experiments also show that the performance of Clasp and of our IDP-system are comparable, IDP performing much better on weight-

[2] ←Encodings, instances and solvers can be found on http://dtai.cs.kuleuven.be/krr.

bounded dominating set, Clasp performing better on magic sequence. For the social golfer problem, the results are indecisive, MZA and Clasp having similar results and number of timeouts. Minisat09z clearly has an advantage over Minisat2.0b, as is to be expected from earlier results.

## 8 Related and future work

Both in the domains of Constraint Programming (CP) and Answer Set Programming (ASP), research on aggregates is an important focus. Within CP, specialized constraints are developed for aggregates, e.g. [**?**]. Within ASP, aggregates occur in language constructs like choice rules, cardinality constraints [14] and weight constraints [6]. Within the SAT domain, Eén et al. [4] have shown how to reduce pseudo-Boolean constraints (of the form $lwr \leq sum\{S\}$, with $S$ a weighted set) directly into SAT. Currently, we are not aware of any work towards supporting aggregates in SMT.

We have also developed a separate $T$-solver to extend FO with inductive definitions (FO(ID)). In the case of aggregates occurring inside the bodies of inductive definitions, we developed a scheme of interaction between both modules to allow satisfiability checking of such formulas, while still maintaining clear separation of concerns. Part of this work has been published in [7], namely the propagation rules for aggregates and an initial implementation of the concepts. We extend this to more general aggregates (using the concept of (anti-)monotonous literals), provide more efficient propagation and present how to implement it as a separate DPLL(T) module.

In the future, we will look to extending existing SMT-solvers with our aggregate module and include other SMT modules. We will improve the solver by implementing the presented watched-literal scheme and include native support for global constraints. We will also use the aggregate module as a basis for developing optimization inference techniques for FO(.).

## 9 Conclusion

In this paper, we presented the logic FO(Agg), an extension of first-order logic with general aggregate expressions. FO(Agg) has interesting monotonicity properties, which allow propagation rules for satisfiability checking which can be efficiently implemented. Algorithms are introduced to extend a DPLL-based search procedure with aggregates within the DPLL($T$)-architecture and implementation and efficiency concerns are addressed. We present implementation optimizations like set-reuse and a watched literal scheme. Several experiments show that our solver can compete with the best ASP solver at the moment on aggregate problems and show a clear advantage compared to encoding aggregates in FO.

## References


1. Alexander Aiken, Dexter Kozen, Moshe Y. Vardi, and Edward L. Wimmers. The complexity of set constraints. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *CSL*, volume 832 of *LNCS*, pages 1–17. Springer, 1993.
2. Second answer set competition. http://dtai.cs.kuleuven.be/events/ASP-competition/index.shtml/.
3. Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczyński. The second Answer Set Programming competition. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *LPNMR*, volume 5753 of *LNCS*, pages 637–654. Springer, 2009.
4. Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.
5. Herbert B. Enderton. *A Mathematical Introduction To Logic*. Academic Press, second edition, 2001.
6. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. On the implementation of weight constraint rules in conflict-driven asp solvers. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 250–264. Springer, 2009.
7. Maarten Mariën. *Model Generation for ID-Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, February 2009.
8. David G. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–132, 2005.
9. Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC'01*, pages 530–535. ACM, 2001.
10. Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *LPNMR*, volume 1730 of *LNCS*, pages 317–331. Springer, 1999.
11. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL($T$). *Journal of the ACM*, 53(6):937–977, 2006.
12. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353, 2007.
13. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in constraint handling rules. In Verónica Dahl and Ilkka Niemelä, editors, *ICLP*, volume 4670 of *LNCS*, pages 446–448. Springer, 2007.
14. Tommi Syrjänen. *Logic Programs and Cardinality Constraints: Theory and Practice*. Doctoral dissertation, TKK Dissertations in Information and Computer Science TKK-ICS-D12, Helsinki University of Technology, Faculty of Information and Natural Sciences, Department of Information and Computer Science, Espoo, Finland, 2009.
15. Allen Van Gelder. The well-founded semantics of aggregation. In *PODS*, pages 127–138. ACM Press, 1992.
16. Johan Wittocx. *Finite Domain and Symbolic Inference Methods for Extensions of First-Order Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Mai 2010.
17. Johan Wittocx, Maarten Mariën, and Marc Denecker. GIDL: A grounder for $FO^+$. In Maurice Pagnucco and Michael Thielscher, editors, *NMR*, pages 189–198. University of New South Wales, 2008.